Apr 91

Final Report on Complete Tool Set for Developing Large Scale Signal Processing Applications on Multicomputers

N00014-90-J-1939

March 1, 1990 to March 31, 1991

Submitted to:
Office of Naval Research
Arlington, Virginia 22217

by:
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The research resulted in a parallel program generator called ASSIGN for building coarse-grained DSP applications modeled as flowgraphs on iWarp. ASSIGN is described in [9, 8]. The underlying theory and partitioning algorithm are detailed in [7]. ASSIGN was was an essential first step that has already proven useful. General Electric used ASSIGN to build a sonar adaptive beam interpolation application on iWarp and Intel has agreed to support ASSIGN as a product in Q2 1992.

Overview

ASSIGN is a tool for building large-scale applications on distributed-memory multicomputers. The programming model is the coarse-grained *flowgraph*, where nodes correspond to coarse-grained sequential computations and directed edges correspond to FIFO queues[2, 5]. Given a flowgraph, ASSIGN automatically generates a parallel program consisting of a C program for each cell (processor) in the target machine. Details such as partitioning the flowgraph, placing and routing the partitioned flowgraph, creating communication pathways, transferring data from cell to cell, flow control, and balancing workload across processors and communication pathways are all handled automatically.

ASSIGN is intended for applications that can be described with static coarse-grained flowgraphs. In particular, it is designed for large-scale digital signal processing (DSP) applications, which are described very naturally using coarse-grained flowgraphs.

Aside from the obvious merits of automating machine-dependent details, the primary virtues of ASSIGN include the hierarchical nature of its flowgraphs, the ability to parameterize both flowgraphs and individual signal processing operations, the ability to model and compile large graphs with thousands of nodes, a flexible and general-purpose mechanism for transferring data to and from the host, and a minimum of run-time scheduling overhead. The primary limitation is that it is suitable only for those applications that can be modeled as static, coarsed-grained flowgraphs.

Flowgraphs

The flowgraph is the underlying programming model for ASSIGN. Under the guise of various names such as block diagrams[10], large-grained dataflow graphs[2], synchronous dataflow graphs[5], and signal flow graphs[11], flowgraphs have been used for years in the literature and in practice to model DSP applications.

A flowgraph is a collection of *nodes* and directed *edges*. Nodes represent computations and edges represent FIFO queues. Each node iterates an infinite number of times. Each iteration, the node consumes some data items from each of its input edges, performs a computation on the inputs, and produces some data items on each of its output edges. The sizes of the inputs and outputs are indicated by integer edge labels.

Figure 1 shows an example flowgraph for an application that performs the same operations on two independent data streams. The edge labels correspond to the number of data items produced and consumed each time a node is

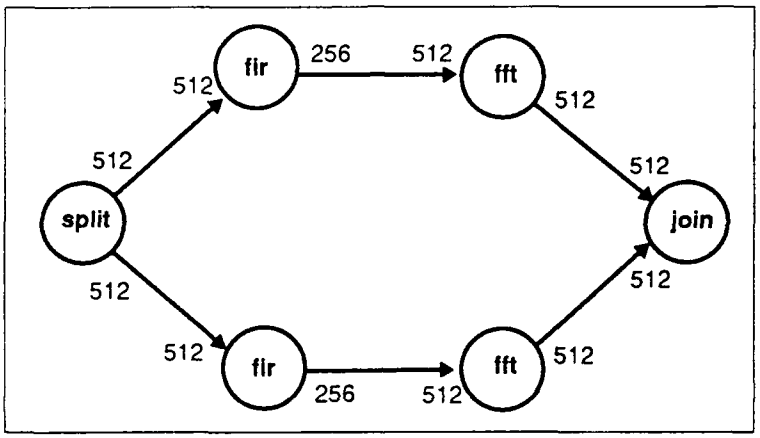

Figure 1: Example flowgraph. Edge labels denote the number of data items produced and consumed each time a node is invoked.

invoked. For example, the node labeled *fir* corresponds to an FIR filter that inputs 512 data items, performs a 2:1 downsampling operation, and outputs the result. The FFT is invoked whenever 512 data points have arrived. Thus, for each invocation of the FFT, there are two invocations of the FIR filter.

ASSIGN flowgraphs are hierarchical. A node in a graph can correspond to a subgraph, so that graphs can be constructed hierarchically out of smaller graphs.

Mapping Algorithm

Assign uses the two-phase mapping algorithm shown in Figure 2. The first phase, partitioning, uses an extension of the chain mapping methods developed in [7] to partition the M nodes of the flowgraph into at most P blocks, where P is the number of the cells in the target machine. Since the throughput of the system is bounded from above by the cell with the maximum load, the optimization goal of this phase is to find a partition that minimizes the maximum load on any particular cell.

There are several interesting aspects of the partitioning algorithm. First, the load model for each cell incorporates communications cost, and thus the optimal partition does not always use all of the cells. Second, although the partitioning problem is intractable in general, the extended chain method we are using is guaranteed to find an optimal partition in polynomial time if the input flowgraph is a chain [7].

The second phase, placement and routing, places the partition blocks from the first phase on the cells of the target array, and routes any exposed edges through the cells of the array. The optimization goal of this phase is to minimize the number of flowgraph edges shared by any particular physical communications link. The placement and routing algorithm is borrowed directly from [6].

The interesting aspect of the placement and routing phase is the simple and beautiful correspondence between exposed graph edges and iWarp pathways. Each pathway is set up once when the program is loaded, and data flow through intermediate cells transparently, with no effect on any computations on the intermediate cells.

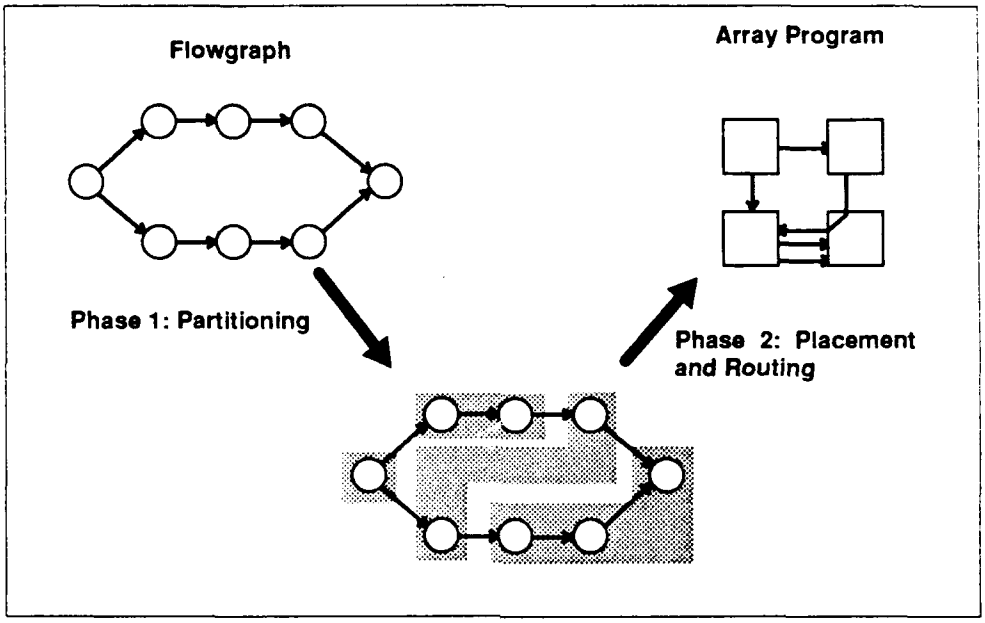

Figure 2: The Assign mapping algorithm. The partitioning phase attempts to minimize the maximum load on any particular cell. The placement and routing phase attempts to minimize the number of flowgraph edges shared by any particular physical communication link.

System Overview

ASSIGN is a front-end to the existing tools supplied by the vendor of a parallel machine. ASSIGN's place in the iWarp tool chain is shown in Figure 3.

Flowgraphs are created by running a C program called a *graph generator*, which uses a library of ASSIGN functions to build and emit a flowgraph on the standard output.

ASSIGN inputs the flowgraph produced by the graph generator along with a collection of *state structures*, which, as we shall see, are arbitrary user–defined C structures that can be used to parameterize an application and hold internal state. On the command line, the user can specify such options as the number of cells to be compiled onto, an optimization level that controls the quality and running time of the mapping algorithm, and the *arbitrary* 2D topology of the target iWarp array.

The output of Assign is a C program that will, at the discretion of the user, be compiled to run on either the Unix workstation or the iWarp array.

Programming Overview

The process of programming applications using ASSIGN is determined largely by (1) the ability to compile flowgraphs onto either the host Unix workstation or the attached parallel computer, and (2) the hierarchical nature of the flowgraphs. The ability to compile onto the workstation allows us to do most of our testing and debugging on the workstation rather on the parallel machine. The hierarchical nature of flowgraphs allows us to build applications in a sequence of incremental steps:

Step 1: Write node functions. These are sequential C functions, instances of which will correspond to nodes in the flowgraph. Each of these functions can be tested independently on the Unix workstation.

Step 2: Generate and test subgraphs. ASSIGN flowgraphs are hierarchical, meaning that graphs can be formed by composing smaller subgraphs. Typically, we independently generate and test a collection of smaller subgraphs on the

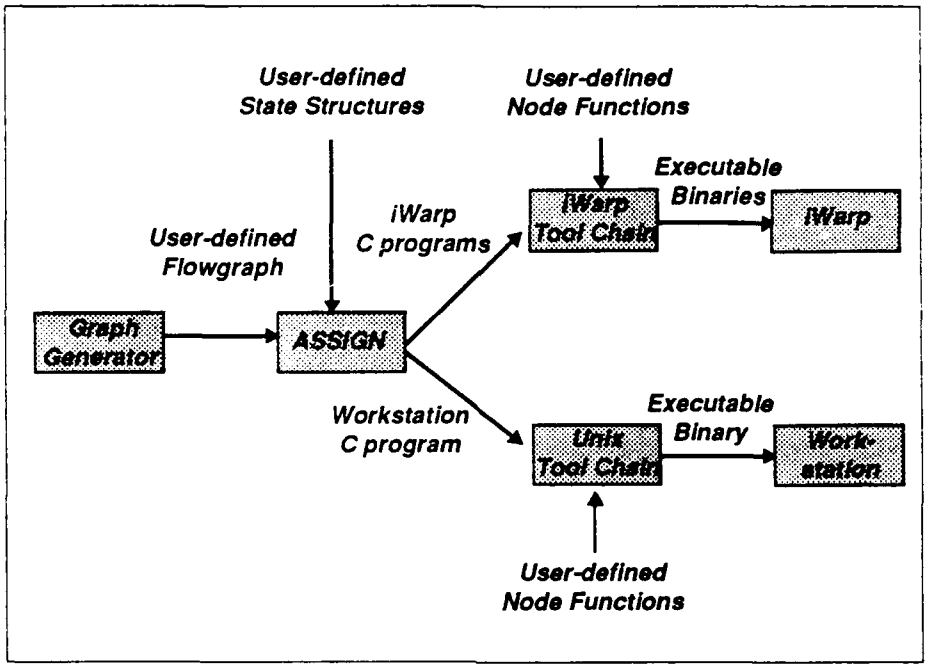

Figure 3: Assign system overview. ASSIGN is a front-end to existing tools. Programs can be compiled onto a parallel computer or a Unix host.

Unix workstation, using the node functions developed in the previous step. As we shall see, flowgraphs are generated by writing and running a C program called a *graph generator*.

Step 3: Combine and test subgraphs. Once we are satisfied that each of the subgraphs is producing correct results, we combine them, *without change*, to form the final graph. The testing of the final graph can be performed entirely on the Unix workstation.

Step 4: Run final graph on parallel machine. When we are satisfied that the final graph is producing correct results, we can recompile the graph, *without change*, and then run it on the parallel machine. It is important to note that most of the program development is done on the Unix workstation and not on the parallel machine.

Related Work

ASSIGN continues the tradition of building parallel program generators for restricted classes of applications. Examples of program generators for image processing, matrix computations, and nested loop computations can be found in [3, 16, 12], respectively.

The flowgraph model has been studied extensively. Some seminal theory on scheduling flowgraphs and an extensive survey of previous results can be found in [5].

Several systems for compiling flowgraph descriptions of signal processing applications have been developed in recent years, including Gabriel[4] at Berkeley and ZC[11] at Carnegie Mellon. Gabriel is a task-level parallel program generator that was originally designed to generate code on single processor systems or small shared memory parallel systems. However, there has been recent work to extend it to distributed-memory systems[15] as well. The ZC system is designed to compile flowgraphs onto linear systolic arrays like the Warp computer [1]. Unlike ASSIGN or Gabriel, where nodes correspond to sequential tasks, ZC allows for nodes to correspond to parallel tasks.

References

[1] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[2] R. G. Babb. Parallel processing with large grain data flow techniques. *Computer*, 17, July 1984.

[3] L. G. Hamey, J. Webb, and I. C. Wu. Low-level vision on Warp and the Apply programming model. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*. Kluwer Academic Publishers, 1987.

[4] E. A. Lee, W. Ho, E. Goei, J. Bier, and S. Bhattacharyya. Gabriel: A design environment for DSP. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(11):1751–1762, November 1989.

[5] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[6] O. Menzilcioglu, H. T. Kung, and S. W. Song. Comprehensive evaluation of a two-dimensional configurable array. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, pages 93–100. IEEE, 1989.

[7] D. M. Nicol and D. R. O'Hallaron. Efficient algorithms for mapping pipelined and parallel computations. *IEEE Transactions on Computers*, 40(3):295–306, March 1991.

[8] D. R. O'Hallaron. ASSIGN: A task-level parallel program generator. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[9] D. R. O'Hallaron. The Assign parallel programming generator. Technical Report CMU-CS-91-141, Carnegie Mellon University, 1991.

[10] A. Oppenheim, A. Willsky, and I. Young. *Signals and Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[11] H. Printz, H. T. Kung, T. Mummert, and P. Scherer. Automatic mapping of large signal processing systems to a parallel machine. In *Proceedings of SPIE Symposium, Real-Time Signal Processing XI*, San Diego, CA, August 1989. Society of Photo-Optical Instrumentation Engineers.

[12] H. B. Ribas. Automatic generation of systolic programs from nested loops. Technical Report CMU-CS-90-143, Department of Computer Science, Carnegie-Mellon University, June 1990.

[13] S. Borkar et. al. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, Kissimmee, FL, November 1988.

[14] S. Borkar et. al. Supporting systolic and memory communication in iWarp. In *17th Annual International Symposium on Computer Architecture*. IEEE Computer Society and ACM, May 1990.

[15] G. C. Sih and E. A. Lee. Scheduling to account for interprocessor communication within interconnection-constrained processor networks. In *Proceedings of the 1988 International Conference on Parallel Processing*, August 1990.

[16] P. S. Tseng. A parallelizing compiler for distributed memory parallel computers. Technical Report CMU-CS-89-148, Department of Computer Science, Carnegie-Mellon University, May 1989.